LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-204894

Out-of-core Interactive Display of Large Meshes Using an Oriented Bounding Box-based Hardware Depth Query

H. Ha, B. Gregorski, K. I. Joy

June 24, 2004

IASTED Computer Graphics and International Conference
Koloa, HI, United States
August 17, 2004 through August 19, 2004

Out-of-core Interactive Display of Large Meshes Using an Oriented Bounding Box-based Hardware Depth Query

Haeyoung Ha
Visualization and Computer Graphics Group
Department of Computer Science, University of California Davis
Davis, CA, USA
email: haeyoungha@sbcglobal.net
Benjamin F. Gregorski and Kenneth I. Joy
Visualization and Computer Graphics Group
Department of Computer Science, University of California Davis
Davis, CA, USA
email: {bfgregorski,kijoy}@ucdavis.edu

ABSTRACT
In this paper we present an occlusion culling method that uses hardware-based depth queries on oriented bounding boxes to cull unseen geometric primitives efficiently. An out-of-core design enables this method to interactively display data sets that are too large to fit into main memory. During a preprocessing phase, a spatial subdivision (such as an octree or BSP tree) of a given data set is constructed where, for each node, an oriented bounding box containing mesh primitives is computed using principal component analysis (PCA). At runtime, the tree indicated by the spatial subdivision is traversed in front-to-back order, and only nodes that are determined to be visible, based on a hardware accelerated depth query, are rendered.

KEY WORDS
Large Triangle Meshes, Interactive Rendering, Occlusion Culling, Hardware Acceleration Methods

1 INTRODUCTION

Large data sets consisting of several million polygons from satellite imagery and laser range scanners are becoming commonplace, and more efficient means to visualize them are necessary. Multiresolution and data simplification methods are often used to reduce the number of primitives sent to the graphics hardware allowing the display of large data sets at interactive rates. Another approach is to reduce the graphics primitive count using occlusion culling and to leverage the performance benefits of retained mode to render the visible regions at full resolution. Occlusion culling reduces the number of primitives sent to the graphics hardware by identifying and ignoring parts of the mesh being visualized that are not visible from a current view point. This approach is very effective for models with high depth complexity.

Our occlusion culling method uses the hardware depth query available on modern graphics hardware and oriented bounding boxes to cull unseen geometric primitives that are within the viewing frustum. An out-of-core design enables data sets that are too large to fit into main memory to be displayed at interactive rates. In a preprocessing phase, we build a hierarchical spatial subdivision of a given dataset. Each node of the hierarchy is associated with a tight-fitting oriented bounding box around the mesh primitives that intersect the node. At runtime, the subdivision tree is traversed in front-to-back order, and a hardware depth query is performed using the oriented bounding box at each node to determine visibility for the node's geometry. Only nodes that are determined to be visible are traversed and rendered.

Contributions The main contributions of our paper are the spatial subdivision of a data set using tight-fitting oriented bounding boxes to perform occlusion culling on large polygonal models. Additionally, we present an out-of-core design which allows models that are too large to fit into main memory to be viewed at interactive rates. Lastly, we leverage the capabilities of modern graphics hardware through the use of hardware-based occlusion queries (AGP) memory to perform fast occlusion tests using the bounding boxes and asynchronous rendering on mesh primitives.

2 PREVIOUS WORK

Much research has been done in occlusion culling and visibility computation. Many culling algorithms have been designed for specialized environments [1] including architectural models and urban data sets composed of large occluders. Occluders are objects in the data set known to cover or occlude parts of the data set (such as walls in architectural visualization). These specialized occluder-based approaches work well for the intended environments but perform less optimally in general settings. Hillesland *et al.* [7] use hardware-based depth query for occlusion culling. They precompute a spatial subdivision of a mesh and render geometric primitives in front-to-back order, using hardware depth queries to determine visibility. For fast front-to-back traversal of a subdivision hierarchy, they use a user-programmable vertex shader. For the power-plant model, which has significant depth complexity, they obtain a speed-up factor of four, on average, over view-frustum culling alone and up to a speed-up factor of ten on constrained cases. Yoon *et al.* [10] describe a system for view-dependent rendering from continuous level-of-detail models with conservative occlusion culling. A vertex hierarchy based on edge collapses is built for level-of-detail rendering, and a cluster hierarchy based on vertex clustering is layered on top of the vertex hierarchy for occlusion. For occlusion, frame-to-frame coherence is exploited by reusing the visible set from the previous frame to approximate the new visible set. Govindaraju et al. [5] show how to generate sharp shadows for complex polygonal environments using hardware occlusion queries.

El-Sana *et al.* [3] integrate occlusion culling with view-dependent rendering. They combine geometric simplification and occlusion culling to achieve a greater reduction in rendered triangles. They do this by adding "visibility" as another parameter in selection criteria of the appropriate LOD. Andújar *et al.* [2] also show an approach that integrates occlusion culling with view-dependent

rendering using hardly visible sets (HVS). These are subsets of the potentially visible cells that contribute only a small number pixels to the overall image. A framework using a user-specified error bound selects a mesh from a fixed set of LOD representations based on the HVS error estimates. Greene *et al.* [6] describe a visibility culling method that uses two hierarchical data structures, an object-space octree and an image-space Z-pyramid, to rapidly cull hidden geometry. They divide the mesh geometry into octree nodes, and scan convert the faces of each node to determine visibility. Thus, if the node is "hidden" none of its children are considered. A Z-pyramid hierarchy is employed to reduce the cost of computing node visibility. Zhang *et al.* [11] uses a hierarchical occlusion map (HOM) for software visibility culling. For each frame, occluders are selected from an occluder database and rendered, forming the occlusion map hierarchy. Occluders are rendered as white polygons on a black background and a depth estimation buffer is constructed to record depth values. The bounding volume hierarchy of the model database is traversed to determine visibility culling.

Our algorithm is similar to the work described in [7] in that a hardware depth query is used for occlusion culling. Our algorithm differs from [7] by using hardware depth query on tight-fitting oriented bounding boxes to cull more occluded geometric primitives. In addition, our method's out-of-core approach (integrated with fast AGP memory) allows data sets that are too large to fit in main memory to be interactively displayed. Similar to Yoon et al. [10] we split clusters using principle component analysis; however, we utilize space partitioning structures (octree and BSP tree) to divide the vertices into clusters. Additionaly, our preprocessing clustering algorithm operates out-of-core, and our runtime viewing algorithm allows models to be viewed at interactive rates in full resolution.

3 Preprocessing Phase

The preprocessing phase begins by computing a spatial subdivision of the model to be visualized. The model consists of vertices and triangles defining the connectivity of the vertices. The mesh is recursively partitioned into smaller volumes and an oriented bounding box that closely fits the geometry contained within each volume is computed using Principal Component Analysis (PCA). A detailed description of PCA is provided in [9]. Oriented bounding boxes computed using PCA are also described by Gottschalk et al. [4] for fast hierarchical collision detection.

The tree defined by the spatial subdivision and the geometry contained inside each node are stored in separate files for use during runtime. The first file, the subdivision file, stores the spatial subdivision and the oriented bounding box corresponding to each node in the tree. The second file, the data file, stores the mesh geometry, grouped according to subdivision tree node, in the same order as the nodes appear in the subdivision file.

The subdivision file consists of an array of tree nodes. Each node contains the following:

- **Child Identifier**. The index of the first child node.
- **Offset**. The disk offset into the data file.
- **U, V, W and Origin**. Three orthogonal axes and origin representing the local frame of the oriented bounding box.
- **Width, Height, and Depth**. Scalar dimensions of the oriented bounding box.

Only leaf nodes have associated geometry so the offset is only used if the node is a leaf node. Similarly, the child index is only used if the node has child nodes. The U, V, and W basis vectors and the origin together represent the local frame that describes the oriented bounding box for a node. Each non-leaf node always has the maximum number of children allowed. Thus, at runtime, this file can be accessed quickly by using the appropriate offsets. The data file contains the mesh geometry for each leaf node in the subdivision tree. Each leaf node has its own local set of vertices and triangles stored as an index face list. Using local sets of geometry allows the indices of a triangle to be stored in two bytes(unsigned shorts) instead of four bytes(unsigned ints). This reduces both the disk space and main memory usage of the geometry. The following information is stored for each leaf node in the subdivision tree:

- Number of vertices and triangles.
- List of vertices (triples of floating-point values) and triangles (triples of indices into the list of vertices).

The subdivision tree is constructed according to a set of user-specified parameters. These parameters are: available memory, maximum tree depth, and maximum number of triangles per leaf node. The maximum tree depth and the maximum number of triangles serve as termination criteria for the tree construction process. We have tested both octree and binary space partition (BSP) tree implementations of the subdivision tree.

Each octree node is represented physically by an oriented bounding box and is subdivided at a split point. Choosing a good split point location is important since it determines how the triangles are distributed among each of the eight children. Even distribution is desired so that the resulting tree is balanced. The average of the vertices contained within a node determines the split point used to compute the child nodes.

A binary space partition is obtained by recursively bisecting bounding boxes with a plane. Each BSP tree node stores its split plane and has two children. Each child represents the space "in front of" or "behind" the split plane. The split plane is chosen to be the plane with the dominant vector of the three principal directions as its normal.

4 Runtime Phase

At runtime, the subdivision file is used to recreate the subdivision tree, which is traversed hierarchically in front-to-back order based upon the current view point. For each node visited, its oriented bounding box is tested for visibility using a hardware depth query. If the node is not visible, traversal is terminated and none of the leaf nodes are rendered, thus, saving the time required to render these elements.

For a given view point, beginning at the root node, we determine in which octant (or half space) the view point lies. For a BSP tree, the plane used is the split plane and front-to-back traversal is simply a variant of an in-order binary tree traversal. For an octree, three planes are tested to determine the traversal order of the children.

For large data sets that contain millions of triangles, it is not possible to load the entire triangle mesh into memory let alone render all of the triangles in a reasonable amount of time. A memory management scheme is used to allocate the available memory for storing node geometry in main memory. Our algorithm uses a combination of fast Accelerated Graphics Port (AGP) memory and slower main memory for storing and rendering node geometry. The AGP supports fast, asynchronous access to memory for transferring data to the graphics card independent of the CPU. Rendering performance is enhanced by allocating a chunk of AGP memory and dividing this into smaller chunks to hold node geometry. Additional main memory is allocated as needed for holding node geometry when the AGP memory store has been exhausted. Two chunks of AGP memory, large enough for rendering geometry from any node, called *reserve* chunks are set aside for use when AGP memory is filled.

This allows us to exploit frame-to-frame coherence between consecutive viewpoints by keeping geometry from recently rendered nodes cached in AGP or main memory.

At the start of each frame, the subdivision tree is traversed from the root node in front-to-back order and the oriented bounding boxes are used to determine the visible regions which are subsequently rendered. The front-to-back tree traversal and rendering of visible geometry incrementally builds a depth buffer that is used for subsequent occlusion queries. If there are no memory chunks available (either AGP or main), the reserve chunks are used for loading and rendering the geometry but are not assigned to the node. Furthermore, memory chunks, corresponding to nodes that have not been visible for several frames, are reclaimed and added to the pool of available chunks. This is implemented using a counter for each memory chunk that is incremented every frame its corresponding node is not visible. If this counter reaches some user-specified threshold, the memory chunk is released from the node and marked as available. The counter is set to zero if the node becomes visible.

An alternative method to this occlusion strategy is to render all visible nodes from frame i and the viewpoint for frame $i+1$ and use this depth buffer as the starting point for the occlusion queries. The advantage of this method is that it can reduce the number of occlusion queries because the initial depth buffer is a much better approximation of the final one. The disadvantage is that it requires extra bookkeeping to track which nodes have been rendered and which nodes from frame i need occlusion tests since some invisible nodes will be rendered as the object is rotated.

Vertex arrays are used for efficient rendering and reduced function call overhead. The implementation used for the results in this paper uses the NVIDIA extension `glDrawRangeElements` to place vertex arrays in AGP memory and the `glSetFenceNV` and `glFinishFenceNV` extensions for synchronization between the GPU and the CPU. However, we note that future implementations should be based upon the recently released `ARB_vertex_buffer_object` extension.

4.1 Hardware Occlusion Query

The hardware occlusion query determines whether or not rendering a set of primitives would affect the image on the screen. Modern graphics hardware allows multiple occlusion queries to be sent at once and permits other processing to continue while waiting for the results. The occlusion query allows a user to determine whether or not to render based on some defined threshold of modified fragments. An adaptive rendering technique could also be developed that selects a mesh from a hierarchy of meshes based on the amount of occlusion. Our algorithm uses NVIDIA's occlusion query extension and renders all of the contained primitives if one or more pixels are affected and ignores the geometry when no pixel is affected. The occlusion query process has three steps:

1. Disable updates to the color and depth buffers.

2. Render the query geometry. In our algorithm, the query geometry is a node's bounding box.

3. Obtain and process the query results. In our algorithm this includes enabling writes to the color and depth buffers and rendering the geometry if the node is visible.

The results of an occlusion query are not available until the query geometry has finished rasterization. This can result in significant performance degradation if the time between initiating the query and reading the results is not filled with other useful calculations. In our algorithm, we keep the occlusion query pipeline as utilized as possible by submitting multiple occlusion queries in succession before the results from the first queries are read. It should be noted that it is possible to lose some amount of culling by submitting multiple queries at once if some of the queries are dependent on each other. Submitting queries for geometry that overlaps other geometry within a set of queries can potentially lead to a false invisibility test. (False invisibility tests can cause more than the necessary number of triangles to be rendered, but this does not generate cracks or holes in the rendered image.)

5 Results

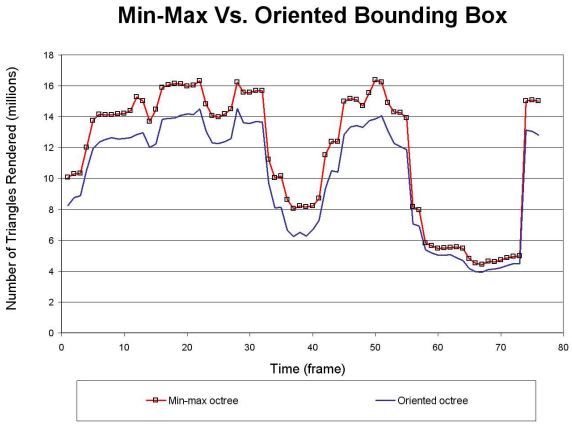


Figure 3: Performance of occlusion culling using an octree constructed with conventional min-max bounding boxes and another octree constructed with oriented bounding boxes. Graphs indicate the number of triangles rendered per frame. Octrees were constructed on the Lucy data set having a maximum of 5000 triangles per leaf node.

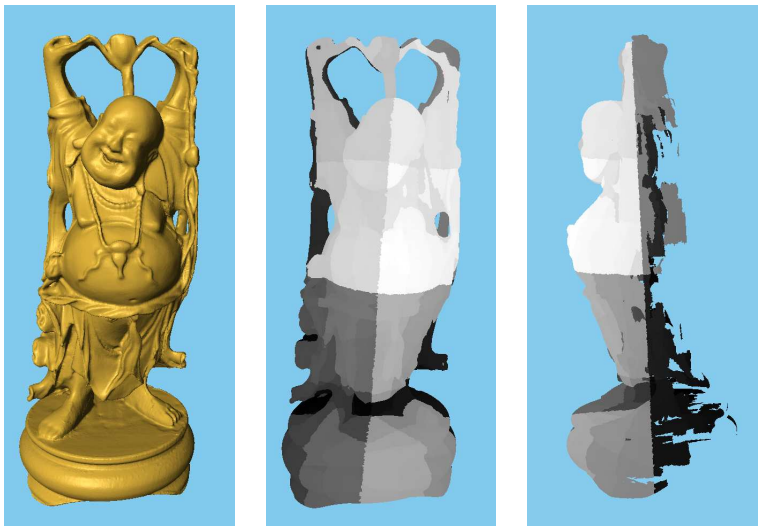
Both an octree and a BSP tree spatial partitions were tested for several large meshes. All data sets were obtained from the Stanford 3D Scanning Repository *(http://www-graphics.stanford.edu/data/3Dscanrep/)*. All tests were performed on a Pentium 4 3.2 GHz system with 2 GB of main memory and NVIDIA's GeForce FX 5200. The Happy Buddha model contains 543K vertices and 1.088 triangles, and the Lucy model contains 14M vertices and 28M triangles. Rendering frame rates ranged from 66 fps to 3 fps. For consistency, a viewing path having 70 to 300 frames was saved for each data set and used to reproduce viewing positions for the different tree implementations and traversal methods. Table 1 shows the average percentages of triangles culled and average frame rates for each data set over the course of this test. Preprocessing requires a few minutes for the Buddha and 30 minutes for the Lucy model (the largest mesh tested).

Figure 1 shows a rendering of the Lucy data set. The left image shows the rendering from the user's view point. At the right, the model is rotated and zoomed to show the back side and the culled regions of the mesh. Frustum culling is automatically included in this method as indicated by the culling of the tip of the torch and the bottom of the torso. Using the octree construction, about 5.66 million triangles (with 80% culling) were rendered in about 0.33 seconds. Figures 2 shows the same for the Buddha data set.

The main feature of this method is the ability to interactively display meshes that are too large to fit into memory. The raw geometry of the Lucy data set requires over 500 MB of memory. This method can interactively display this data set at up to 5 frames per second using an artificial main memory limit of 270 MB. Another feature is the use of tight-fitting oriented bounding boxes for improved

(a) (b) (c)

Figure 1: Rendering of Lucy model using an octree for the spatial partitioning. Image (a) shows a lit, shaded rendering of the Lucy model. Image (b) shows the nodes colored in gray scale based upon tree traversal (white being the closest to the viewer). Image (c) shows a side view of the nodes used to render images (a) and (b).



(a) (b) (c)

Figure 2: Buddha model and rendered nodes using an octree for the spatial partitioning. Image (a) shows a lit, shaded rendering of the Buddha model. Image (b) shows the nodes colored in gray scale based upon tree traversal (white being the closest to the viewer). Image (c) shows a side view of the nodes used to render images (a) and (b).

culling. For the Lucy data set, two octrees were constructed and compared: one using conventional min-max bounding boxes and the other using oriented bounding boxes. Both were constructed using the same parameters and each were tested using the same viewing path. The octree having oriented bounding boxes, on average, culled 6% additional triangles and up to 7.5% additional triangles than the conventional octree, see Table 2. Figure 3 shows a graph of the rendered triangle count obtained using both trees.

The traversal method was tested to see how pipeline stalls affect rendering and culling results. Two variations on traversal were implemented for comparison. Variation (a) –denoted in the graphs with asterisk– performs one hardware depth query at a time, for each node, as the tree is traversed. Variation (b) traverses the tree, keeps track of the sequence of nodes and issues the depth queries in groups of nodes. Figures 4 and 5 show the results for both tree types. Variation (a) culls more triangles but variation (b) achieves faster rendering time. This happens because variation (b) issues multiple queries at once. (Culling rate can be affected by overlapping oriented bounding boxes within a set of queries as previously discussed in Section 4.) Variation (b) achieves faster rendering time due to reduced pipeline stalls since less time is spent on queries.

When using the same parameters for tree construction, an octree has more leaf nodes than a BSP tree due to the fact that an octree split produces four times as many children. Generally, the leaf nodes (in an octree) store fewer triangles, which leads to smaller bounding boxes, ideal for culling. This is indicated by the graphs shown in Figures 4 and 5 since octrees have better culling results. However, the efficiency of the memory management scheme is adversely affected by the greater number of leaf nodes since this number is proportional to the number of memory chunks needed. Since fixed-sized memory chunks (as determined by the largest leaf node) are used and there are more nodes with fewer triangles, less memory is actually utilized. This causes more disk access and increases the rendering time. This explains why the BSP tree culls less geometry than the octree but renders faster. A histogram showing the number of nodes having a certain number of triangles, for both octree and BSP tree constructions for the Lucy data set, is shown in Figure 6. The BSP tree has a better distribution since there is a high concentration of leaf nodes containing a high number of triangles. The octree distribution shows most of the leaf nodes containing a few number of triangles. (The remaining data sets had similar results.)

Data set	Octree		BSP tree	
	Avg. % culled	Avg. frame rate	Avg. % culled	Avg. frame rate
Happy Buddha	49.8%	62.5	41.0%	66.7
Lucy	63.9%	1.9	55.6%	2.3

Table 1: Average percentages of triangles culled and average frame rates for each data set.

Octree type	Min number rendered	Max number rendered	Avg. rendered	Avg. % culled
Oriented	3,585,050	13,896,857	9,471,981	67%
Standard	3,995,501	15,753,649	10,967,944	61%

Table 2: Comparison of two different octree constructions for Lucy data set: one with conventional and the other with oriented bounding boxes. The octree with oriented bounding boxes culls, on average, 1.5 million (6%) additional triangles.

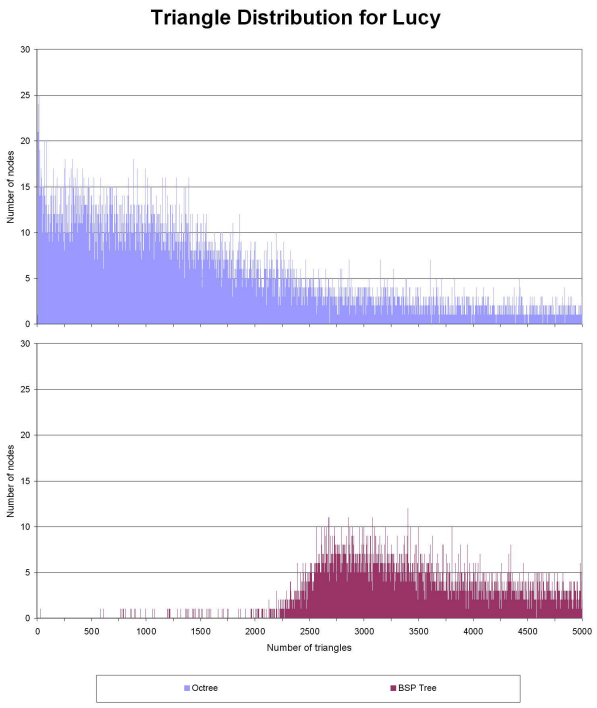


Figure 6: Triangle distribution for Lucy data set. The top image shows the distribution for the octree and the bottom for the BSP tree. This graphs plot the triangle distributions of the leaf nodes for each tree.

6 Conclusions and Future Work

We have presented a new occlusion culling algorithm for rendering large polygonal models at their full resolution. A spatial subdivision of a given data set is computed and traversed in front-to-back order to perform visibility culling. Our approach integrates tight-fitting oriented bounding boxes (computed by PCA) with an out-of-core rendering approach (using fast AGP memory) for a display system that culls a large number of occluded triangles efficiently. We have tested our system on several large models consisting of several million triangles. The Buddha and Dragon data sets, each containing approximately a million triangles, are rendered at 30-66 frames per second, and the Lucy data set, with 28 million triangles, is rendered at 2-3 frames per second. In addition, with an out-of-core design approach, our algorithm is able to interactively display data sets that are too large to fit into main memory.

Our future work is focused on improving the performance and scalability of both phases of our algorithm. The preprocessing phase can take a long time for very large data sets. The current implementation is file I/O intensive due to memory constraints. A more efficient file I/O management scheme will help reduce required preprocessing time. Additionally we would like to investigate bottom-up tree building algorithms based on local clustering and integrated our preprocessing with streaming mesh formats such as those used in [8].

Although a large number of geometric primitives can be culled using this method, in some circumstances, there may still be too many visible regions that need to be rendered. A data simplification method or multiresolution analysis is needed to further reduce the primitive count to maintain interactivity. As hinted at Section 4, an adaptive rendering technique could be developed to select a mesh from a hierarchy of meshes depending on the number of pixels affected or the distance from a view point.

Acknowledgements

This work was supported by the National Science Foundation under contract ACI 9624034 (CAREER Award), through the Large Scientific and Software Data Set Visualization (LSSDSV) program under contract ACI 9982251, and through the National Partnership for Advanced Computational Infrastructure (NPACI); the National Institute of Mental Health and the National Science Foundation under contract NIMH 2 P20 MH60975-06A2; the Lawrence Livermore National Laboratory under ASCI ASAP Level-2 Memorandum Agreement B347878 and under Memorandum Agreement B503159; and the Lawrence Berkeley National Laboratory. We thank the members of the Visualization and Graphics Research Group at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis.

References

[1] J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, pages 41–50, 1990.

[2] C. Andújar, C. Saona-Vázquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of detail through hardly-visible sets. In *Computer Graphics Forum*, volume 19, pages 499–506, August 2000.

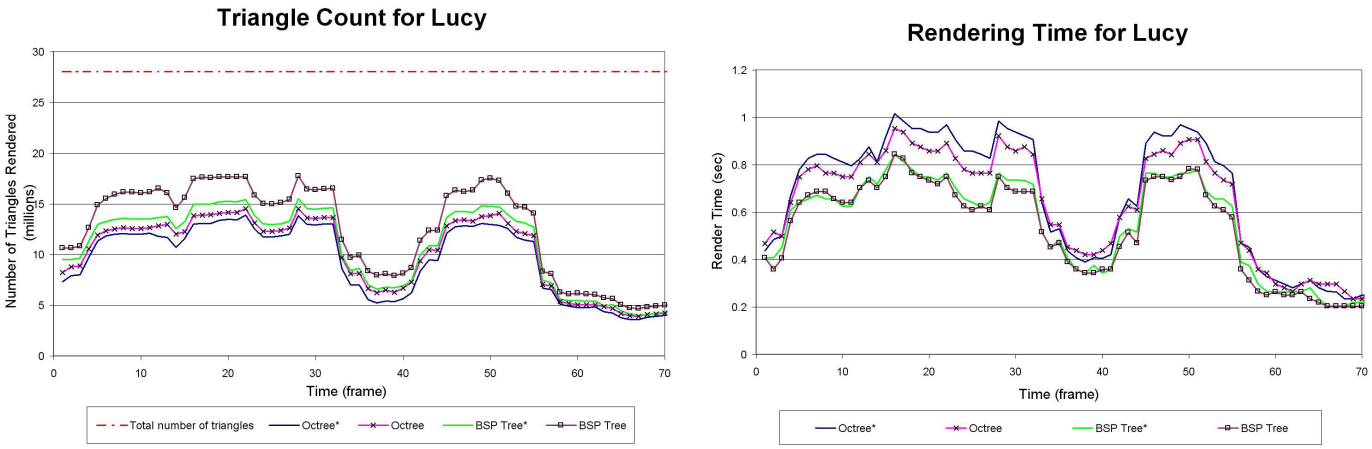


Figure 4: Left: The number of triangles rendered at each frame for the Lucy data set. The octree culls more triangles than the BSP tree. Right: Comparison of the rendering time per frame for the octree and the BSP tree for Lucy data set. Asterisk denotes variation (a) traversal method.

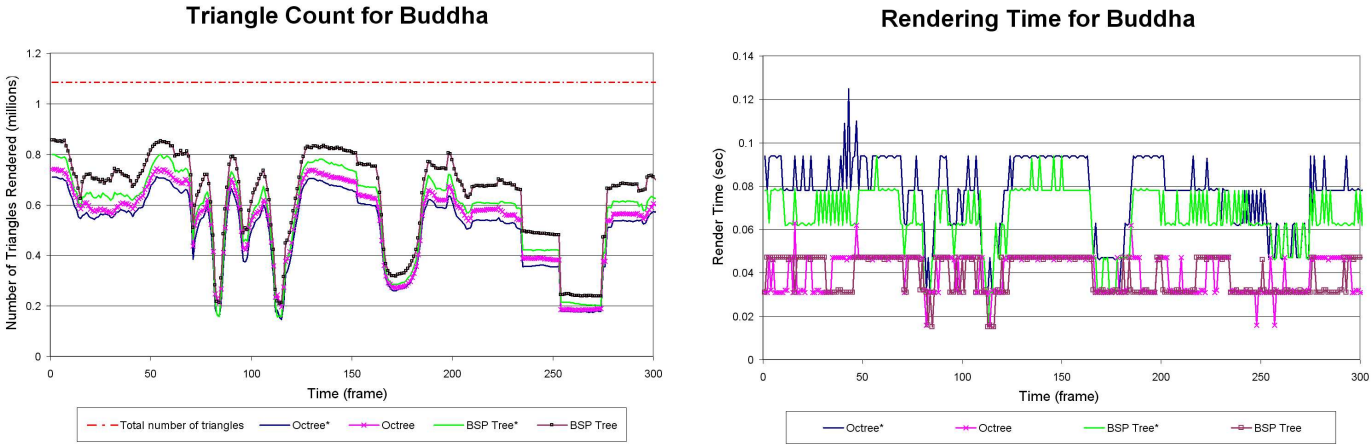


Figure 5: Left: Comparison of the number of triangles rendered for each frame for Buddha data set. Right: Comparison of the rendering time per frame of the octree and BSP tree for Buddha data set. Asterisk denotes variation (a) traversal method.

[3] J. El-Sana, N. Sokolovsky, and C. Silva. Integrating occlusion culling with view-dependent rendering. In *Proceedings of IEEE Visualization*, 2001.

[4] S. Gottschalk, M.C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of SIGGRAPH*, 1996.

[5] Naga K. Govindaraju, Brandon Lloyd, Sung-Eui Yoon, Avneesh Sud, and Dinesh Manocha. Interactive shadow generation in complex environments. In *Proceedings of SIGGRAPH*, pages 501–510, 2003.

[6] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series*, pages 231–240, 1993.

[7] K. Hillesland, B. Salomon, A. Lastra, and D. Manocha. Fast and simple occlusion culling using hardware-based depth queries. Technical Report UNC-CH-TR02-039, Computer Science Department, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, 2002.

[8] Martin Isenburg, Peter Lindstrom, Stefan Gumhold, and Jack Snoeyink. Large mesh simplification using processing sequences. In *Proceedings of IEEE Visualization*, 2003.

[9] I.T. Jolliffe. *Principle Component Analysis*. Springer-Verlag, New York, NY, 1986.

[10] Sung-Eui Yoon, Brian Salomon, and Dinesh Manocha. Interactive view-dependent rendering with conservative occlusion culling in complex environments. In *Proceedings of IEEE Visualization 2003*, 2003.

[11] H. Zhang, D. Manocha, T. Hudson, and K. Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of SIGGRAPH*, pages 77–88, August 1997.

This work was done in part under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.